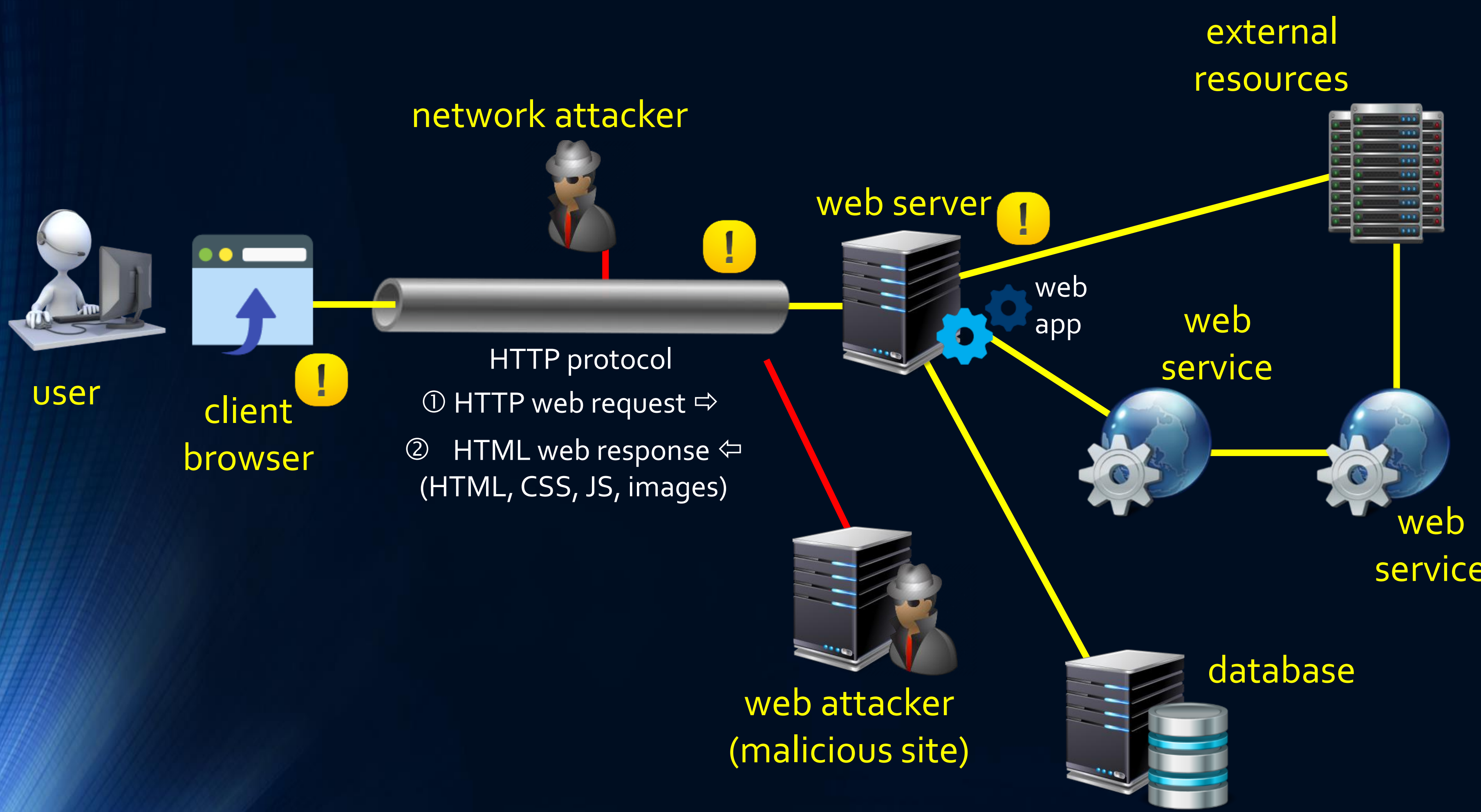
Web Security

COOKIES
OAUTH 2.0 AND OPENID CONNECT
TOKENS
CONNECTION PROTECTION IN SEVERAL FLOWS

APM@FEUP

Web applications

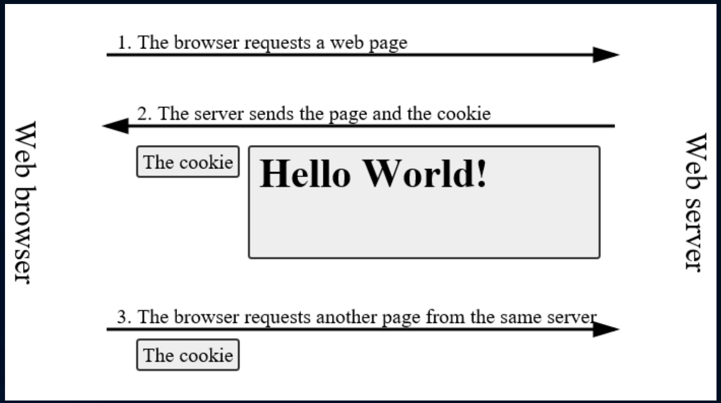


Security needs

- **As any other application and resource access**
  - Web apps often need user identification, authentication, authorization
- **The HTTP protocol is stateless**
  - Some mechanism to assure that several requests come from the same user, after authentication, is needed
  - Establishment of a session
  - Cookies were invented in 1994 (Netscape), patented, and standardized
    - IETF RFC 2109 and RFC 2965, with the more recent RFC 6265 (2011)
    - They are automatically transported between web app and browser
    - They can carry session identification

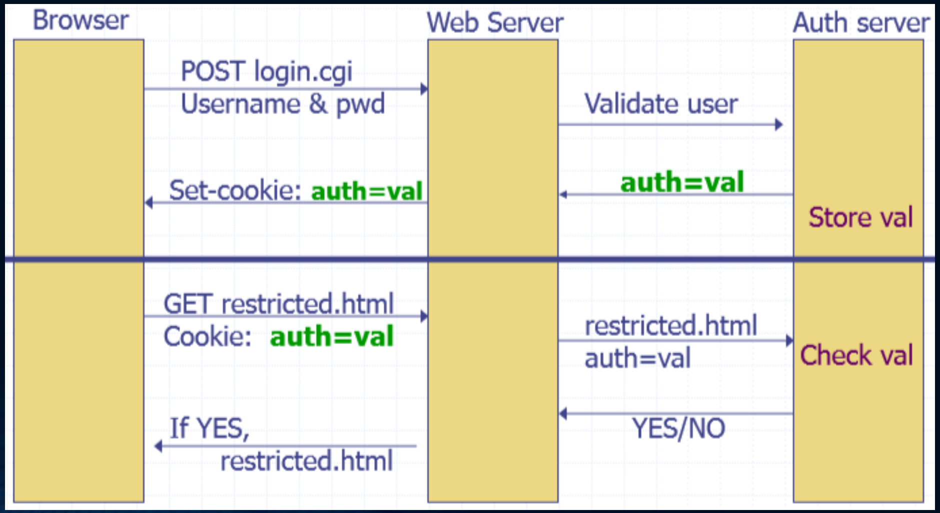
```
HTTP/1.0 200 OK
Content-type: text/html
Set-Cookie: theme=light
Set-Cookie: sessionToken=abc123; Expires=Wed, 09 Jun 2021 10:18:14 GMT
…
```

```
GET /spec.html HTTP/1.1
Host: www.example.org
Cookie: theme=light; sessionToken=abc123
…
```



Cookie authentication

- **Besides a pair name-value cookies can have more attributes**
  - Domain and Path specify the server domain (and subdomains) and the address (and subpages) to where cookies can be returned
  - Expires (or Max-Age) specifies the validity in time
    - If omitted, only valid for the current session
  - Secure and HttpOnly limits the cookie communication to encrypted transmission only (the first) and not readable by client-side scripting (the second)
- **Using some authentication/authorization protocol**



Session hijacking

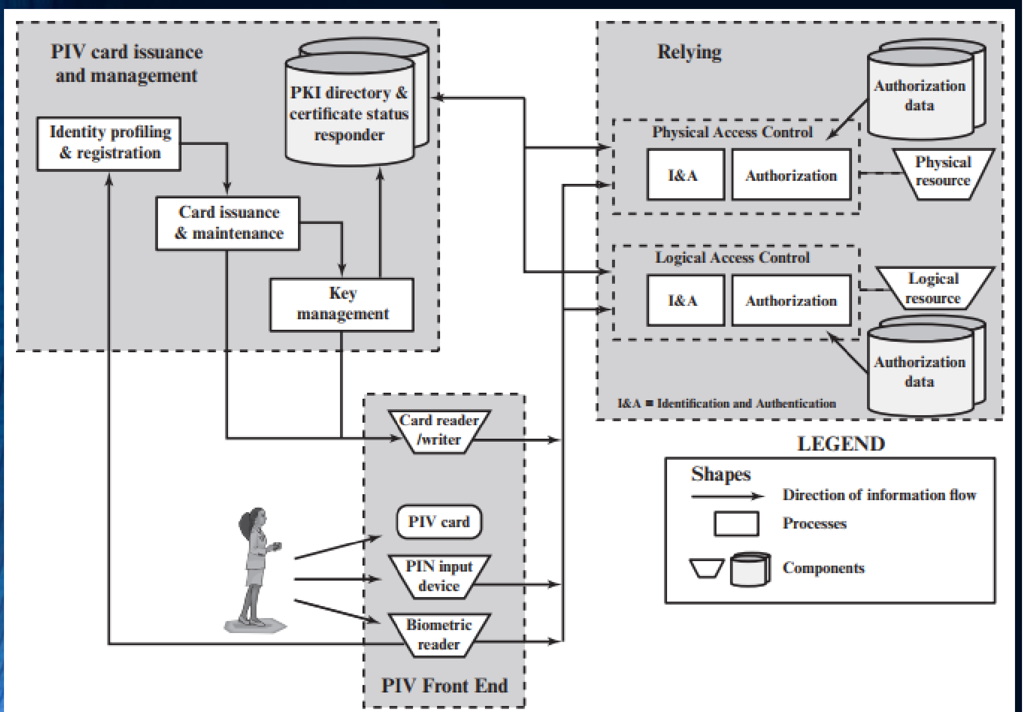
- Cookies can be transmitted in clear text
  - Vulnerable to eavesdropping
  - Once a valid cookie is captured, it can be used directly or used in a man-in-the-middle attack
  - Counter-measure: protect the channel (SSL/TLS with HTTP – HTTPS)
- DNS cache poisoning
  - Fabrication of sub-domains to get the cookies
- Malicious addresses
  - Accessed using cross-site scripting (XSS)
    - Script in the same site directs information to another (malicious) site
  - Performing operations on a legitimate site through cross-site request forgery (CSRF)
    - User executes script in a malicious site that uses non-expired cookies in valid operations on previous visited site
  - Proxy request
    - A proxy server is specified through XSS

Web authentication / authorization

- **Many systems have been proposed and developed**
  - For many general-purpose scenarios
  - Using specialized servers as identity and/or authorization providers
  - They can use external devices to identify the user
    - A PIV system, using a smartcard, and a PIN or biometric 2nd factor
  - In large enterprises, a single authentication server can perform this operation for may web applications
  - Or several organizations can rely on a third-party identification and authentication server
    - These are called single sign-on solutions (or SSO)
  - These web security mechanisms that involve several servers rely on
    - Automatic redirections between them (HTTP 302 (temporary change))
    - Small document for information transport (tokens)

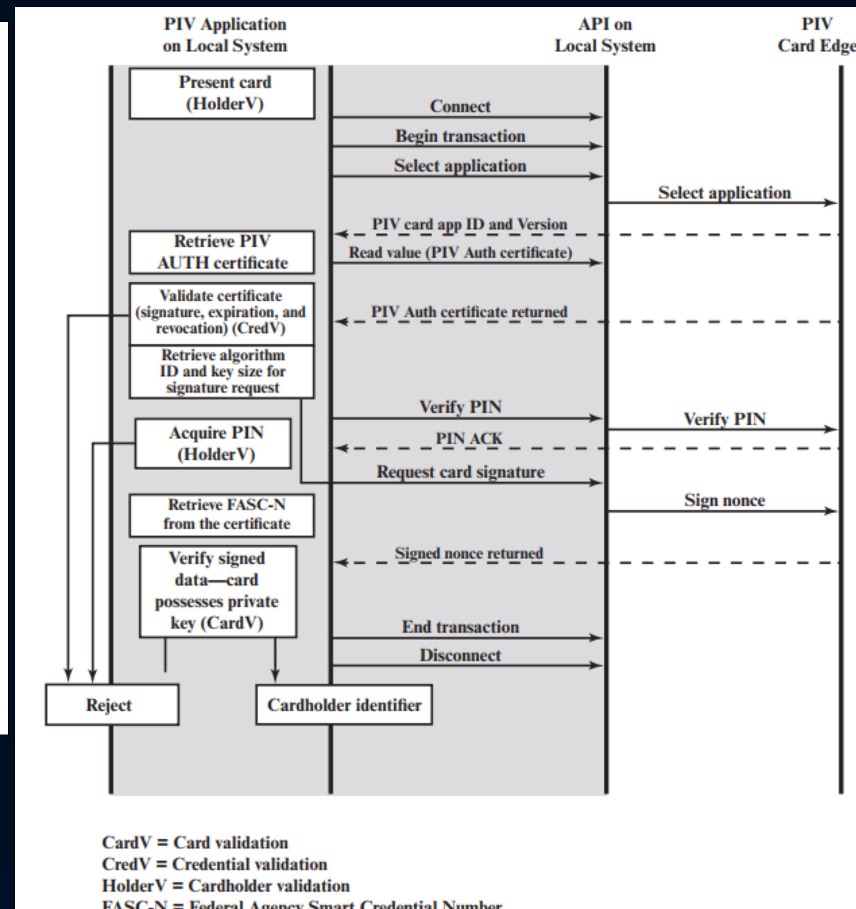
PIV – Personal Identity Verification

- **Based on smartcard possession**
  - Standardized by NIST (FIPS 201-2) / European countries have similar
  - Usually requires 2FA (card + PIN / biometrics)



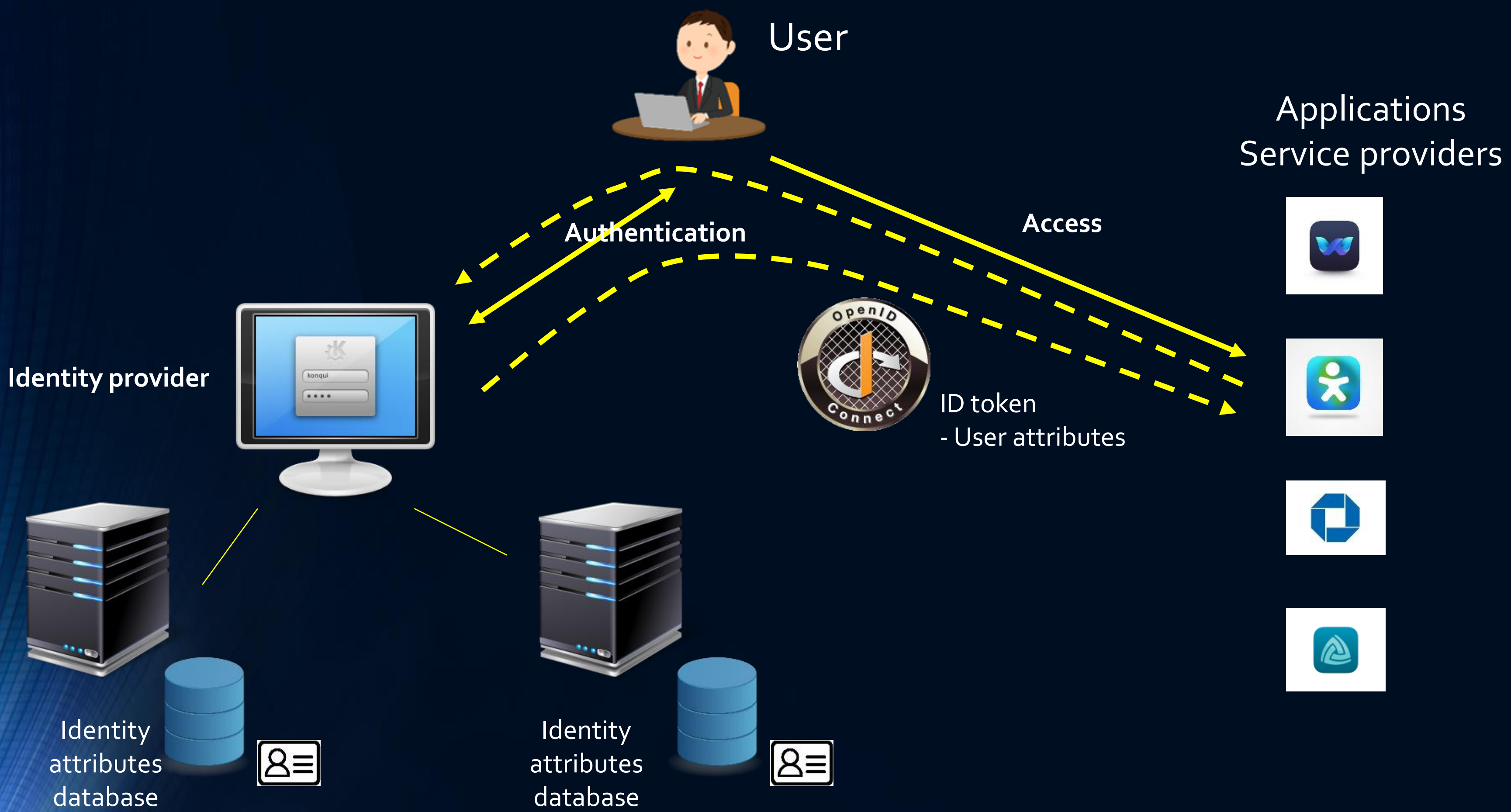
PIV System

- based on a signed certificate
- a signature proving the private key possession matching the certificate



Authentication

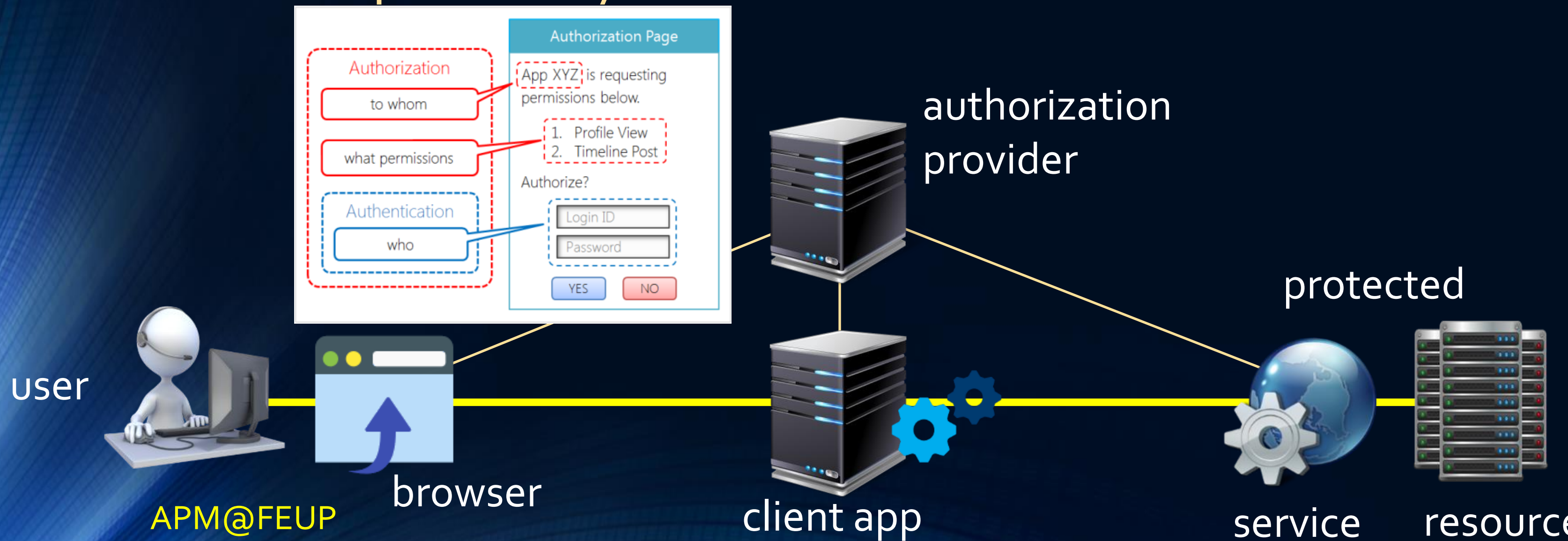
Single-sign-on and federated authentication



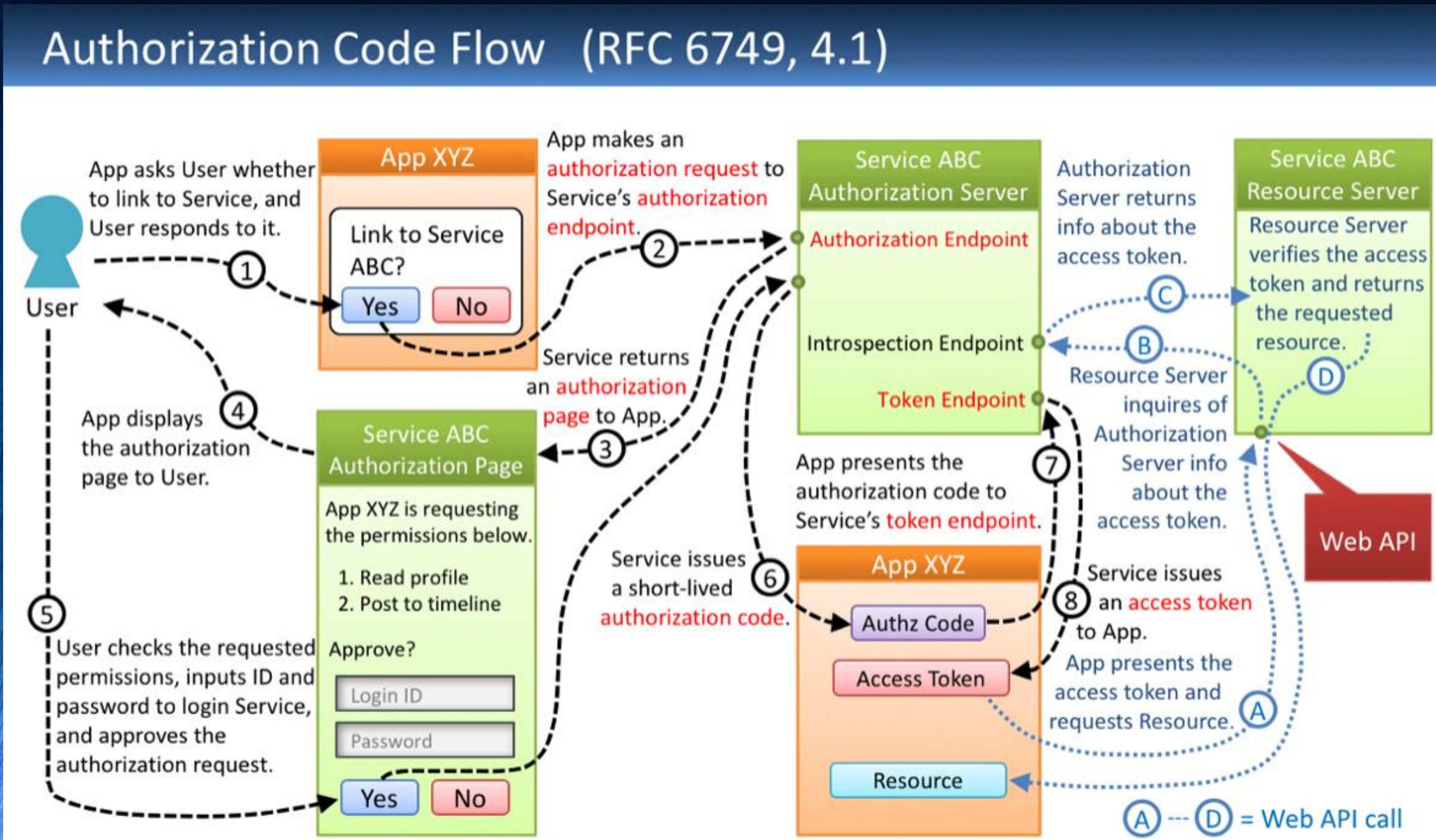
Shibboleth
SAML
OpenId

OAuth 2.0 Authorization Actors

- **OAuth was specified for allowing users be aware of operations in protected resources (usually created by them) by web apps that use the resources**
  - OAuth 2.0 is standardized and described in RFC 6749
    - Specifies an authorization flow for web APIs and resource access on behalf of a web application and user
    - It's not specifically an authentication protocol, but implicitly must include authentication
      - Depends on the quality of the user registration
      - It can be adapted for many situations and scenarios

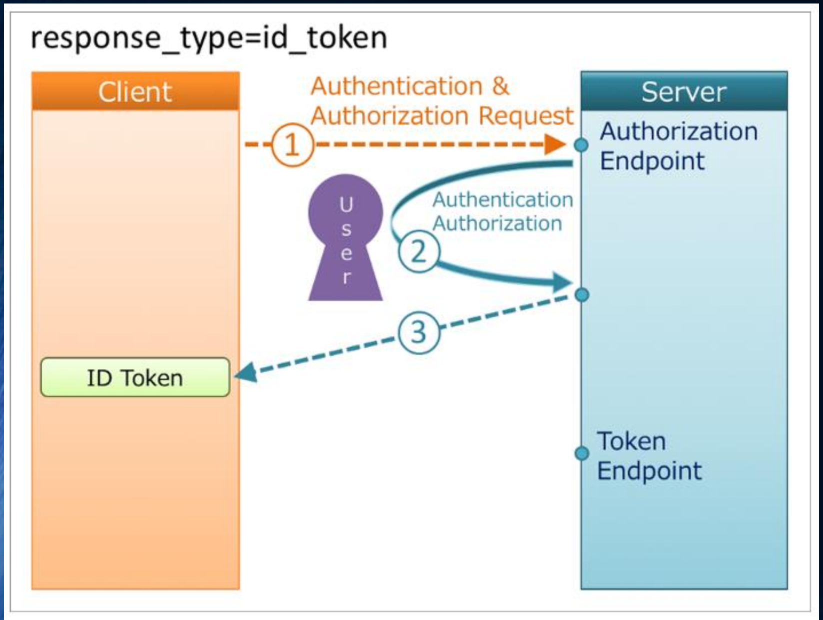


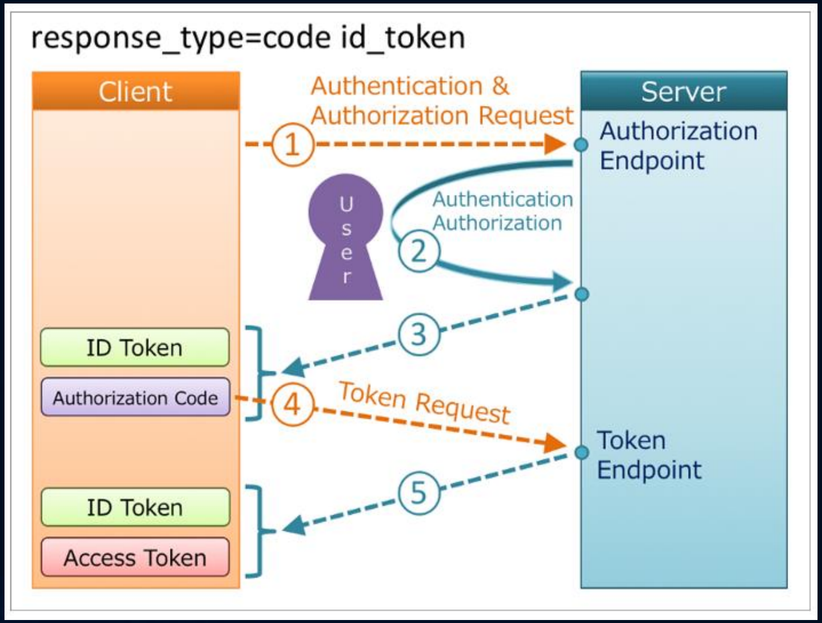
OAuth 2.0 authorization basic flow



OpenID Connect

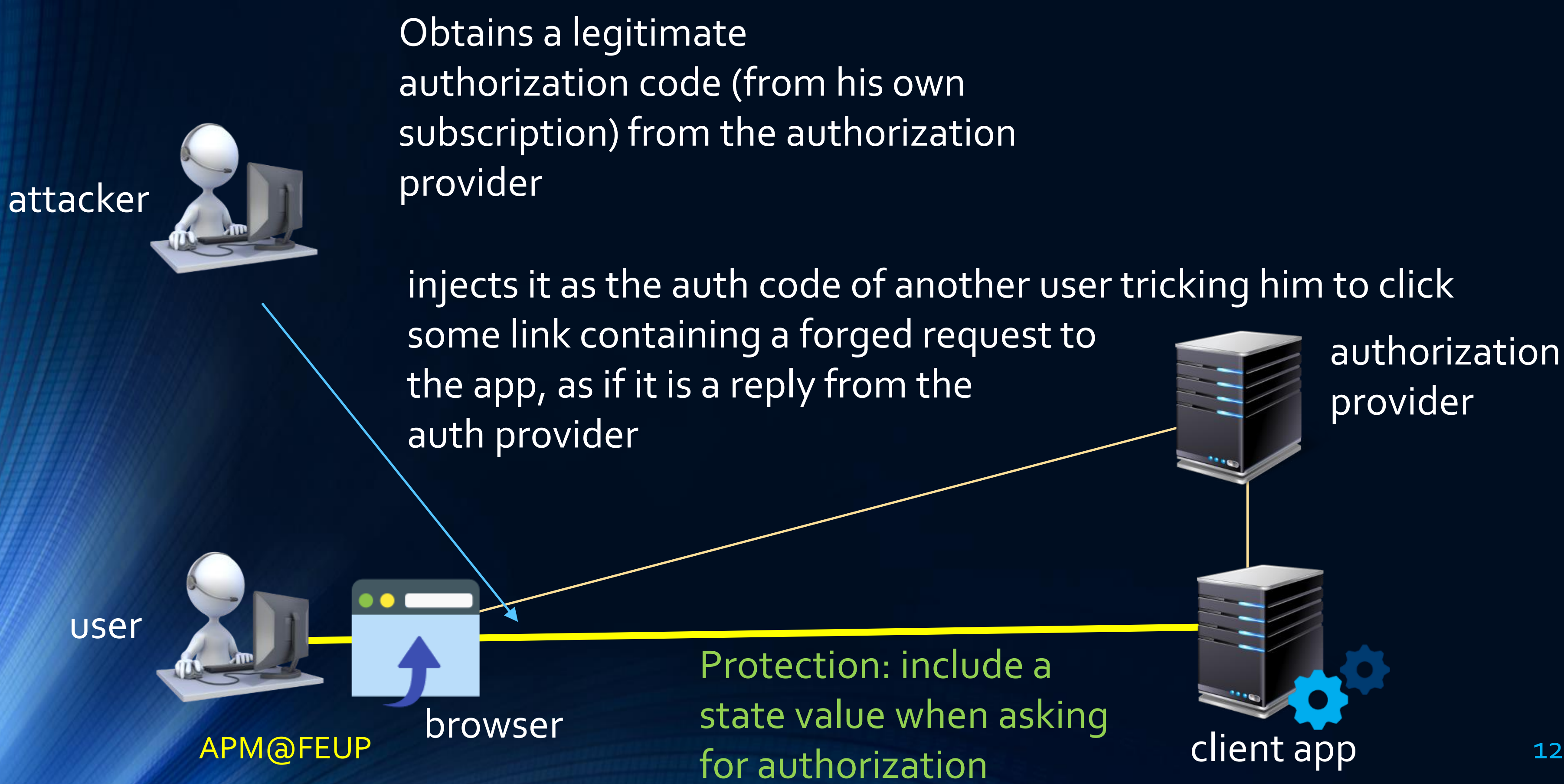
- **OAuth 2.0 does not provide any direct user identification**
  - The web app does know nothing about the user
    - Authorization codes and access tokens are opaque to the app
- **OpenID Connect extends OAuth**
  - Uses provider authentication and supplies an identification token
    - represents the user and contains user info (claims)





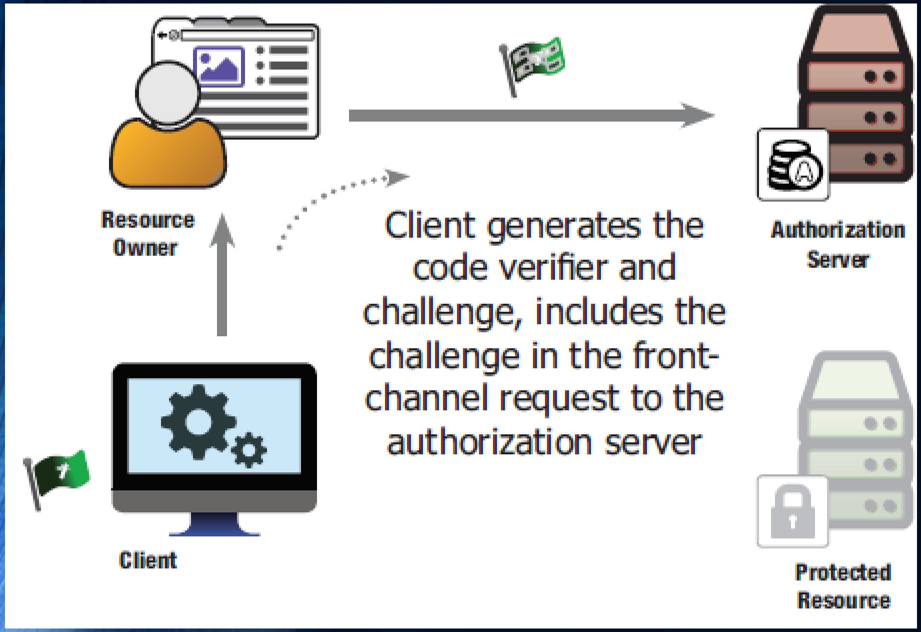
Protective implementation of OAuth

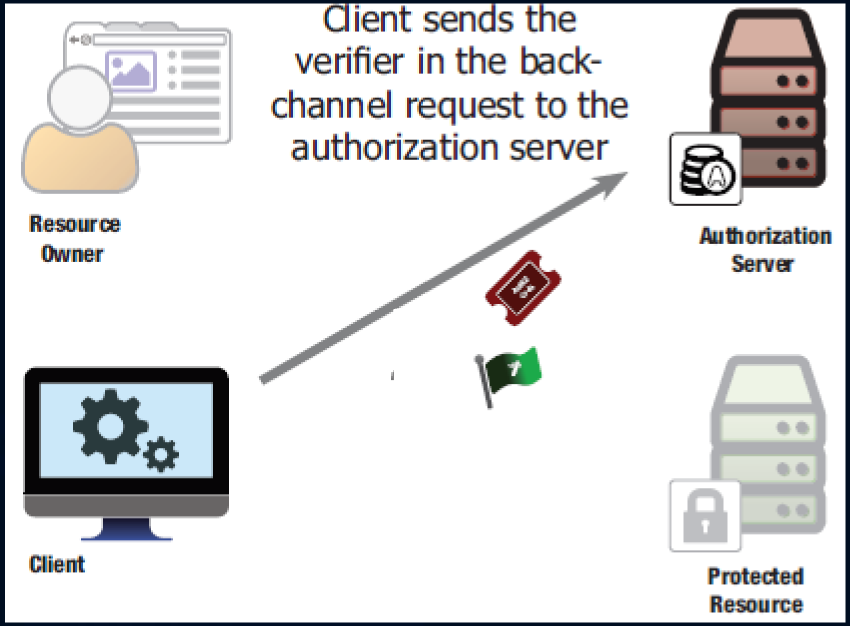
- **RFC 6819 recommends good practices in OAuth 2.0 implementations**
  - All of them should be followed
  - One of them addresses a potential CSRF attack



OAuth code stolen protection

- **User interrupts access after obtaining a valid auth code**
  - Because the auth code comes in a parameter in the redirection from the auth server, it remains in the user's browser history …
  - Potentially an attacker can see it in the browser history, an perform a legitimate authorization replacing his own code with another user code

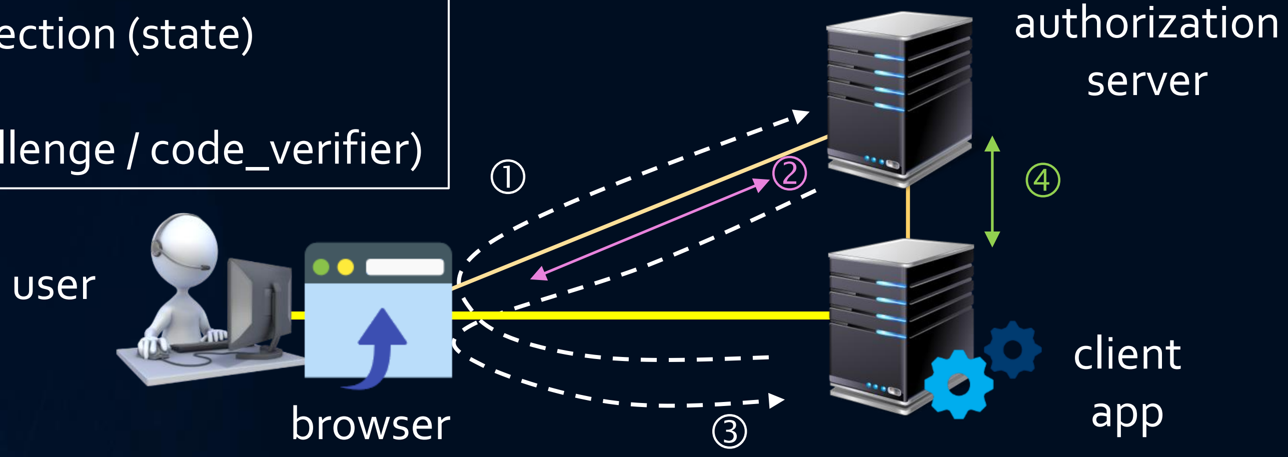




Protection: Proof Key for Code Exchange (PKCE)

OAuth code grant and token exchange

Security protections:
CSRF protection (state)
and PKCE
(code_challenge / code_verifier)



① Authorization request (redirect)

state=xxxxxxxx & code_challenge=yyyyyyy & code_challenge_method=S256

② Authorization dialog (direct)

code_verifier (random) is generated and stored in the client application

code_challenge = H(code_verifier)
code_challenge_method specifies which H

③ Code response (redirect)

state=xxxxxxxx & code=cccccccc (authorization code)

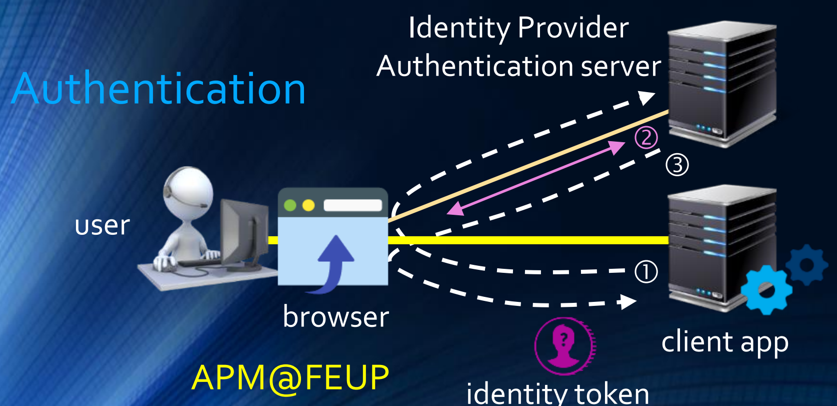
④ Token exchange (direct)

code=cccccccc & code_verifier=zzzzzzzzzz

← The access token is returned if the code is verified

Tokens

- **Tokens are small documents protected against**
  - forgery (usually signed by the originator)
  - disclosure and modification (encrypted and authenticated)
  - The destination (audience) can verify, know the origin, and read the content
- They usually carry authentication, authorization data, user identity
  - In the form of name/value pairs, aka claims
  - The audience trusts the issuer (IdP, AuthN or AuthZ services)
- Tokens can use a JSON format (called 'jots', aka as standard JWT)
  - RFC 7519, together with RFC 7515 (JWS), RFC 7516 (JWE), RFC 7517 (JWA), RFC7518 (JWK)
  - Used together these standards form the JOSE (JSON Object Signing and Encryption) defined and exemplified in RFC 7165 and RFC 7520



JWT format with a signature (JWS)

- **These tokens carry information directly from an issuer to the audience (the application that uses it)**
  - e.g., an identity token from an IdP to a client app
  - Using a cryptographic signature, the audience can verify the integrity and the origin

The signature is performed over the 2 first parts
can be a HMAC (shared key)
or use RSA or ECC (asymmetric)

{ 'typ': 'JWT',
'alg': 'HS256' }

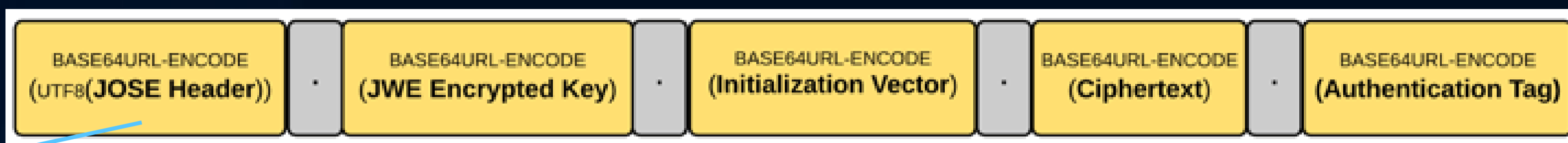
{ 'iss': issuer
'sub': subject
'aud': audience
'exp': expiration
'iat': issued at
'jti': unique id }

'nonce': anti-replay
'acr', 'amr': authn characterization
'at_hash', 'c_hash': companion hashes

OpenId Connect token

JWT with encryption (JWE)

- **When a token contains confidential info, it should use JWE**
  - E.g., when received by an app to be used in a resource server, the app doesn't need to know the content
  - JWE specifies a 5-part token



{ 'typ': 'JWT',
'enc': 'A256GCM'
'alg': 'RSA-OAEP' }

random symmetric key encrypted by an asymmetric public key (from the audience)

random IV
(different for each token)

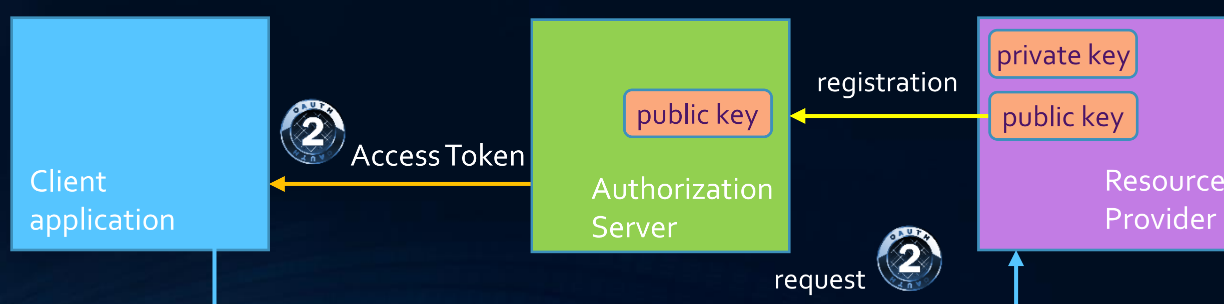
the encrypted payload.

the MAC produced by the GCM algorithm.

The encryption must be performed with a symmetric key in AES with authentication and AD (the GCM mode is the most used). The AD is derived from the header byte sequence.

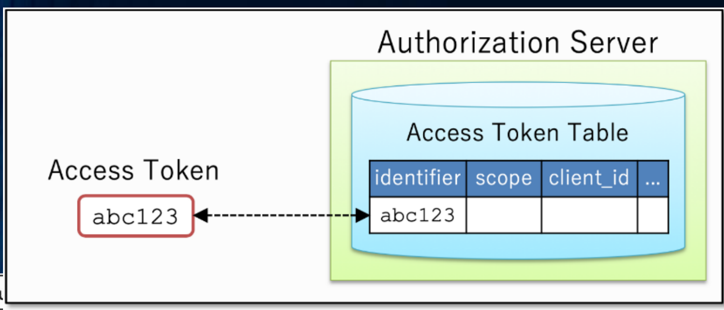
The destination server must be previously registered with the Authorization server and its public key stored.

Sometimes to guaranty to the client app knowledge of the origin of the access token, this JWE can be the payload of a JWS, verified and extracted at the app.

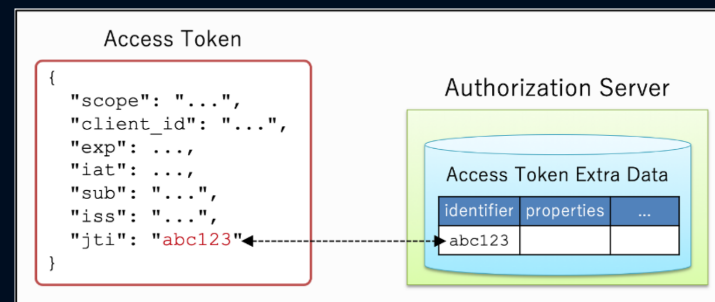


Opaque tokens and introspection

- **These tokens carry on just a meaningless random string**
  - The claims are maintained on a database at the emitter (authorization server for access tokens)
  - The emitter must have an introspection endpoint with an authenticated access to the claims of a token
    - It's also possible a hybrid implementation



Opaque token



Hybrid token

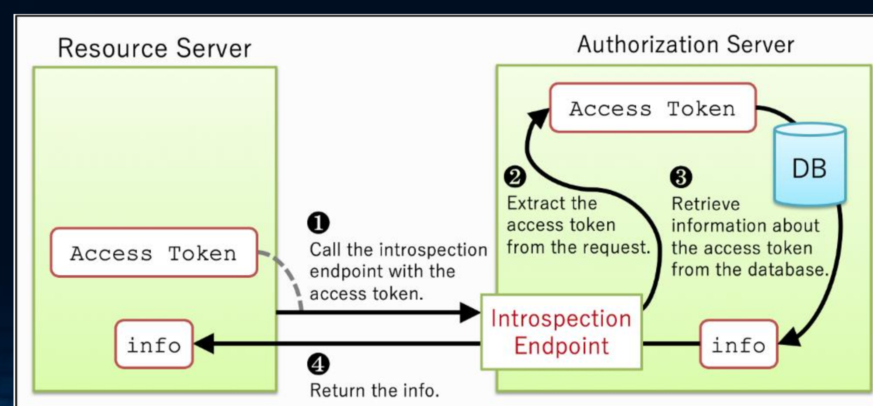
Requ[est]

```
POST /introspect HTTP/1.1
Host: server.example.com
Accept: application/json
Content-Type: application/x-www-form-urlencoded
Authorization: Basic czZCaGRSa3F0MzpnWDFmQmF0M2JW

token=mF_9.B5f-4.1JqM&token_type_hint=access_token
```

Response from Introspection Endpoint (RFC 7662, Section 2.2)

```
HTTP/1.1 200 OK
Content-Type: application/json

{
  "active":          true,
  "client_id":       "l238j323ds-23ij4",
  "username":        "jdoe",
  "scope":           "read write dolphin",
  "sub":             "Z5O3upPC88QrAjx00dis",
  "aud":             "https://protected.example.net/resource",
  "iss":             "https://server.example.com/",
  "exp":             1419356238,
  "iat":             1419350238,
  "extension_field": "twenty-seven"
}
```

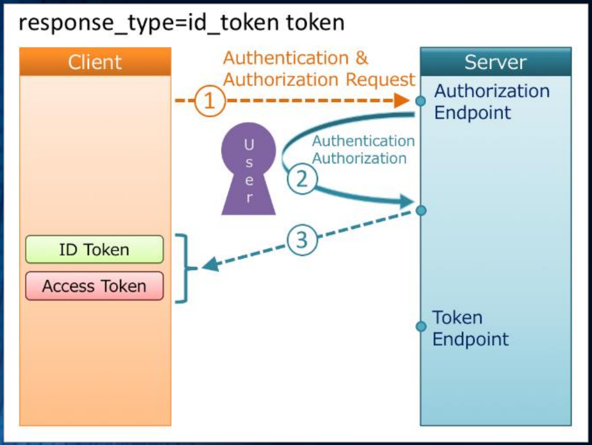
The introspection request



The UserInfo endpoint

➢ From OpenID Connect specification

- **The response from a successful authentication is an IDToken**
  - **It only proves authentication of a user with a given ID**
  - **To obtain user information a request to a user info endpoint must be made with an access token (obtained at the same time)**



The access token should contain the user id in the 'sub' claim and possibly a 'user' or 'username' claim
The 'scope' claim must include "openid"

The UserInfo endpoint of the AuthN/AuthZ server is treated as a Resource endpoint, so the access token is sent in the Authorization header

Request:

```
GET /userinfo HTTP/1.1
Host: server.example.com
Accept: application/json
Authorization: Bearer <access_token>
```

→

Sample response:

```
HTTP/1.1  200 OK
Content-type: application/json

{
  "sub": "9XE3-JI34-00132A",
  "preferred_username": "alice",
  "name": "Alice Smith",
  "email": "alice.smith@example.com",
  "email_verified": true
}
```

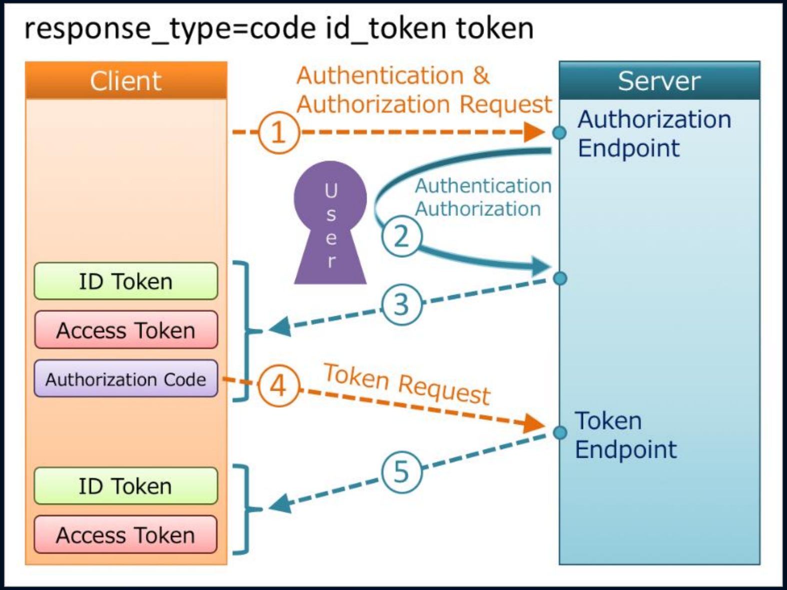
UserInfo and Resource provider access

- **The access token returned by OAuth can grant access**
  - To the UserInfo endpoint on the AuthZ server itself
  - To the Resource provider with the permissions granted to/by the user
- **Sometimes it is desirable to separate**
  - OpenID Connect has a flow allowing that

The access tokens here are different:

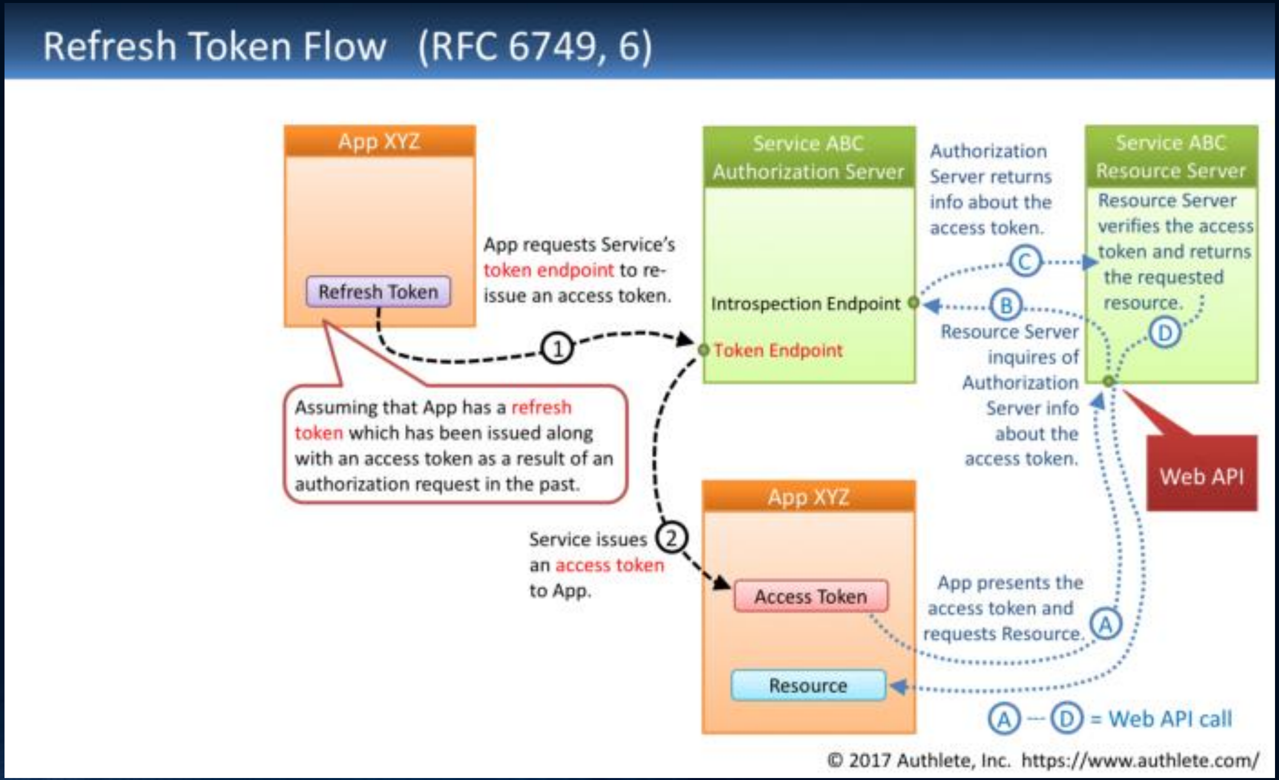
The first can contain only the "openid" scope (and other related defined by the OpenId specification)

The second can contain only the scopes related to the resource provider



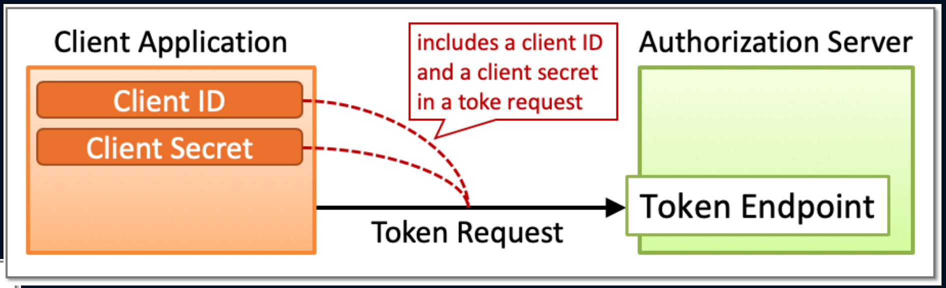
Refresh tokens

- **Access tokens should be very short-lived**
  - A few minutes, allowing only a small number of requests
  - When they expire a new one should be obtained
  - To avoid a new authorization with user intervention, many implementations return a refresh token, together with the access token
  - Refresh tokens live a longer period (like an hour or more)
  - They can be used to get another access token

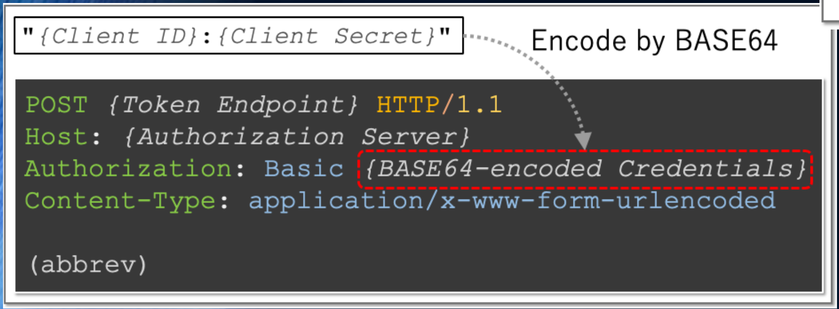


App and resource server authentication

- IdP and AuthZ Servers need to recognize their clients
  - Usually, they need to be registered previously
    - There are standard protocols to register dynamically, or use some OOB way
    - Either way they should be confirmed by an administrator
  - In the registration a unique ID is assigned (e.g., a client_id property) and also a shared secret (client_secret) or a pair of asymmetric keys
  - All requests to AuthN/AuthZ servers must include authentication data

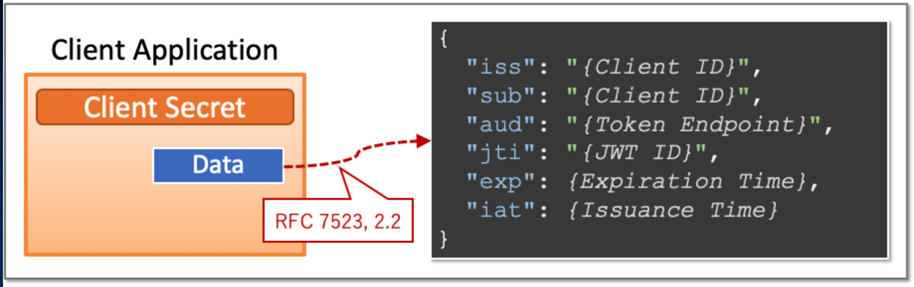


Common form of request authentication (always using TLS)



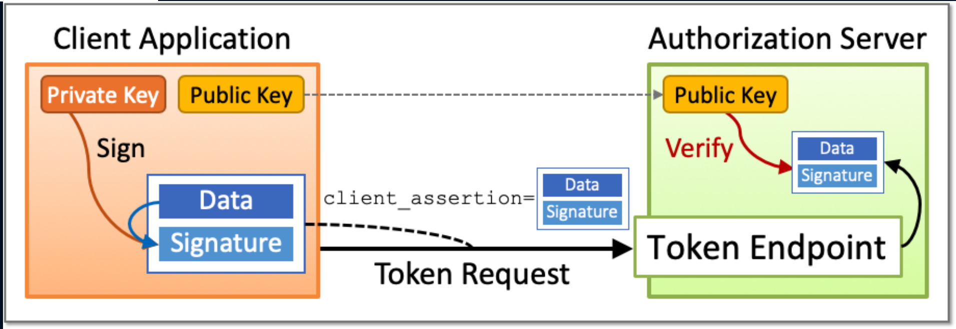
App and resource server authentication (2)

- Another way is using a client assertion



A JSON object is filled with client data

It is signed, sent as a parameter, and verified at the server with A public key established at registration



- The only unauthenticated request accepted should be the initial authorization request (starts the direct dialog with user)

```
GET {Authorization Endpoint}
  ?response_type=code                // - Required
  &client_id={Client ID}             // - Required
  &redirect_uri={Redirect URI}       // - Conditionally required
  &scope={Scopes}                    // - Optional
  &state={Arbitrary String}          // - Recommended
  &code_challenge={Challenge}        // - Optional
  &code_challenge_method={Method}    // - Optional
  HTTP/1.1
HOST: {Authorization Server}
```

Permissions and the scope claim

- Oauth does not specify how to represent permissions
  - It specifies the ‘scope’ claim only as a list of words space-separated

```
"scope" : "email profile"
```

- The ‘scope’ content can be requested by the app in the initial authorization
  - It should be presented to and authorized by the user
  - It should be checked by the AuthZ server, knowing the user and resource server
  - The AuthZ server can grant all or only a subset of the ‘scope’ words
  - It is included in the Token endpoint response, and in the access token
  - It should be checked by the resource provider (it should also know the user)

```
POST {Token Endpoint} HTTP/1.1
Host: {Authorization Server}
Content-Type: application/x-www-form-urlencoded

grant_type=authorization_code  // - Required
&code={Authorization Code}     // - Required
&redirect_uri={Redirect URI}   // - Required if the authorization
                               //   request included 'redirect_uri'.
&code_verifier={Verifier}      // - Required if the authorization
                               //   request included
                               //   'code_challenge'.
```

Request to exchange a code by a token in the /token endpoint
Notice the code_verifier (PKCE) parameter

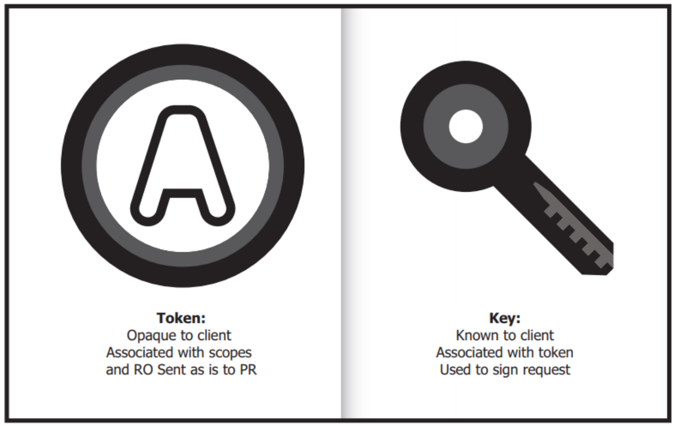
The Client app should also authenticate with the server using one of the previous methods

```
HTTP/1.1 200 OK
Content-Type: application/json;charset=UTF-8
Cache-Control: no-store
Pragma: no-cache

{
  "access_token": "{Access Token}",    // - Always included
  "token_type": "{Token Type}",        // - Always included
  "expires_in": {Lifetime In Seconds}, // - Optional
  "refresh_token": "{Refresh Token}",  // - Optional
  "scope": "{Scopes}"                  // - Mandatory if the granted
                                       //   scopes differ from the
                                       //   requested ones.
}
```

Successful response from the AuthZ server

Bearer vs PoP tokens

- **Client apps present access tokens to a resource provider**
  - Usually in the Authorization header as a Bearer token
  - They are honored by the server (if valid), independently of the sender
  - What if, from a server or app vulnerability, they are stolen?
    - The resource and operation that they grant access, can also be stolen
    - Bearer tokens are like cash, they grant access to who ever have them
  - To protect against this possibility, we can use PoP tokens
    - PoP = proof of possession
  - With this kind of tokens, the resource provider should be able to check that who sends them is the same app that has requested them
    - The AuthZ server associates a key with each token when they are emitted



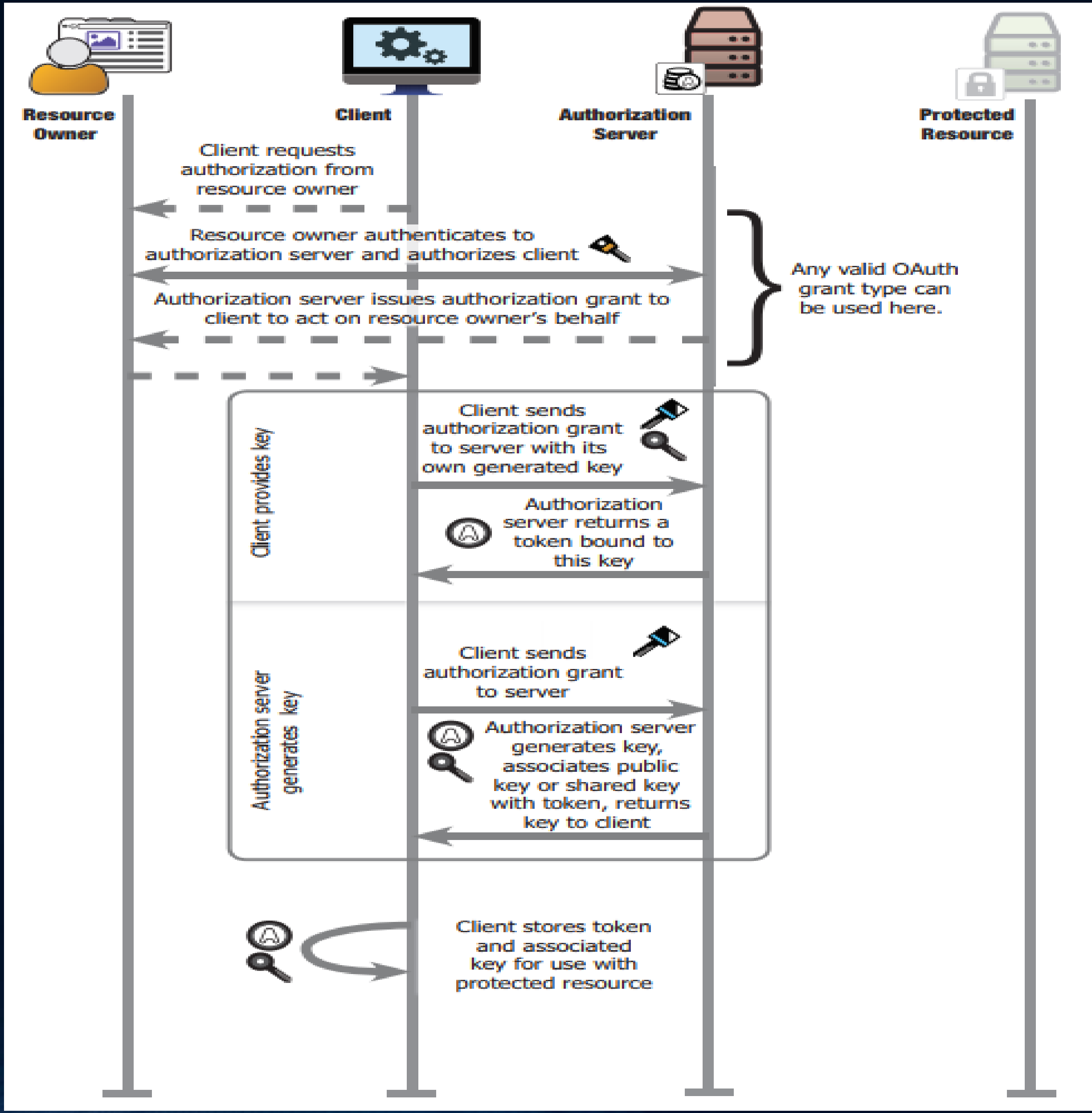
PoP tokens

- **The associated key is generated in the exchange of code**
  - It can be generated in the client or AuthZ server, and can be symmetric or asymmetric

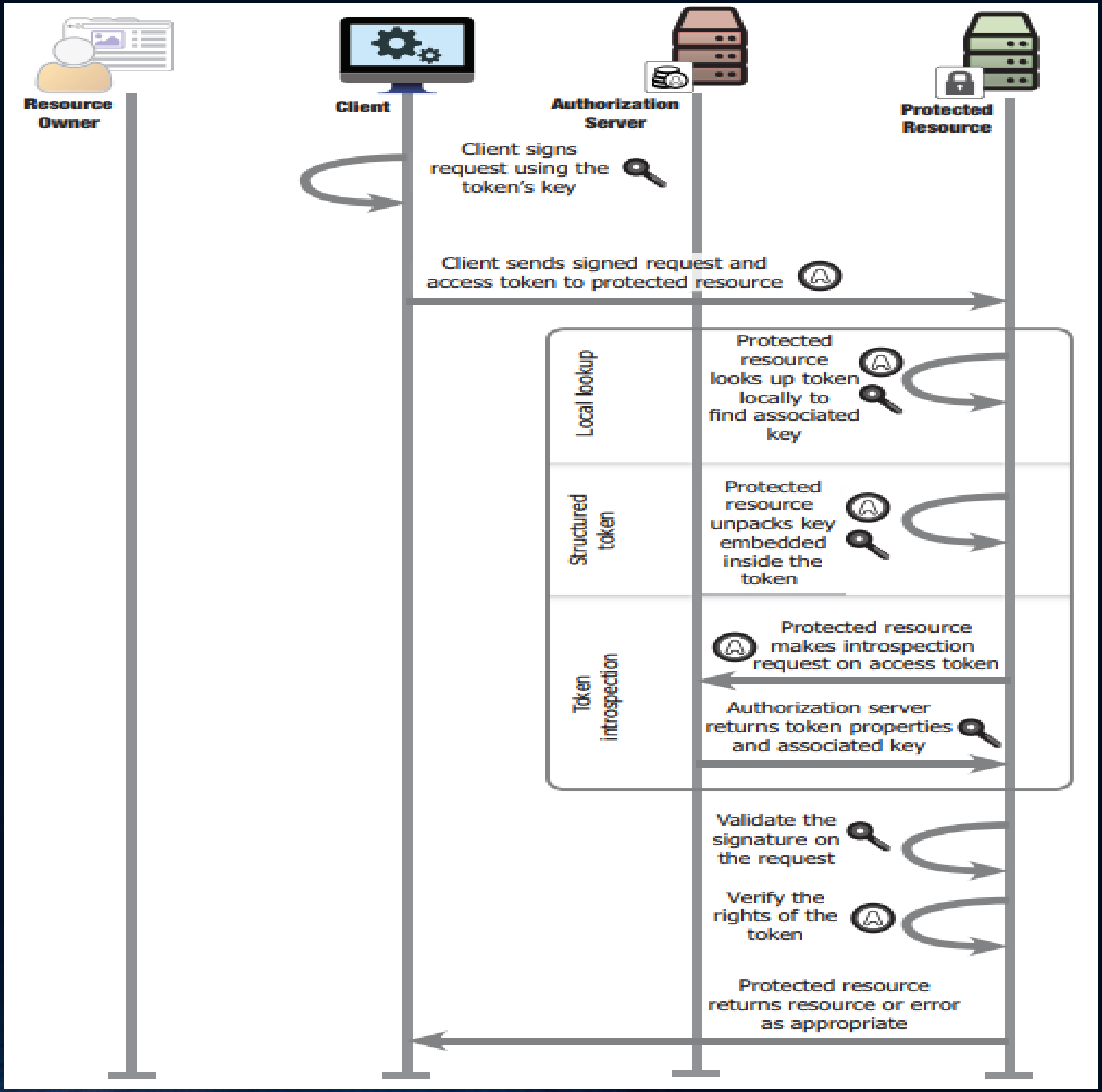
		Provided By:	
		Client	**Server**
Key Type:	**Symmetric**	Not generally a good idea, since the client could be choosing a weak secret, but possible for clients with a Trusted Platform Module or other mechanism capable of generating truly secure shared keys	Good for constrained clients or clients that can't generate secure keys
	Asymmetric	Good for clients that can generate secure keys, minimizes the knowledge of client's private key; client registers public key only, server returns public key only	Good for clients that can't generate secure keys; server generates key pair, returns key pair

  - For a symmetric key both the client and server must know and store it
    - The server can include it inside an encrypted JWT (a JWE)
  - For asymmetric the server stores the public and the client both
    - Again, the server can embed the public key in a JWE

PoP tokens generation phase



PoP tokens use and verification



Response from the token endpoint

- If a PoP token is returned, and the server generated a key or keys, the token endpoint response should include them
- In the token endpoint request and response keys should be transmitted using the JWK specification
  - A JSON object different for each kind of key
  - Example of a response containing a pair of RSA keys
    - These keys are always ephemeral


```
{
  "access_token": "8uyhgt6789049dafsdf234g3",
  "token_type": "PoP",
  "access_token_key": {
    "d": "RE8jjNu7p_fGUcY-aYzeWiQnzsTgIst6N4ljgUALSQmpDDlkziPO2dHcYLgZM28Hs8y
        QRXayDAdkv-qNJsXegJ8MlNuiv70GgRGTOecQqlHFbufTVsE480kkdD-zhdHy9-P9cyDzp
        bEFBOeBtUNX6Wxb3rO-ccXo3M63JZEFSULzkLihz9UUW1yYa4zWu7Nn229UrpPUC7PU7FS
        g4j45BZJ_-mqRZ7gXJ0lObfPSMI79F1vMw2PpG6LOeHM9JWseSPwgEeiUWYIY1y7tUuNo5
        dsuAVboWCiONO4CgK7FByZH7CA7etPZ6aek4N6Cgvs3u3C2sfUrZlGySdAZisQBAQ",
    "e": "AQAB",
    "n": "xaH4c1td1_yLhbmSVB6l-_W3Ei4wGFyMK_sPzn6glTwaGuE5_mEohdElgTQNsSnw7up
        NUx8kJnDuxNFcGVlua6cA5y88TB-27Q9IaeXPSKxSSDUv8nllt_c6JnjJf8SbzLmVqosJ-
        aIu_ZCY8I0w1LIrnOeaFAe2-m9XVzQniR5XHxfAlhngoydqCW7NCgr2K8sXuxFp5lK5s-
        tkCsi2CnEfBMCOOLJE8iSjTEPdjoJKSNro_Q-pWWJDP74h41KIL4yryggdFd-8gi-E6uHE
        wyKYi57cR8uLtspN5sU4110sQX7Z0Otb0pmEMbWyrs5BR3RY8ewajL8SN5UyA0P1XQ",
    "kty": "RSA",
    "kid": "tk-11234"
  },
  "alg": "RS256"
}
```

Client app token preparation

- **The client app creates a JSON object containing**
  - The original token, a time stamp, and some HTTP request data


```
{
  "at": "8uyhgt6789049dafsdf234g3",
  "ts": 3165383,
  "http": { "v": "POST", "u": "locahost:9002" }
}
```


  - Then this is used as a payload in JWS token, signed with the symmetric or private key, corresponding with the association in the AuthZ server

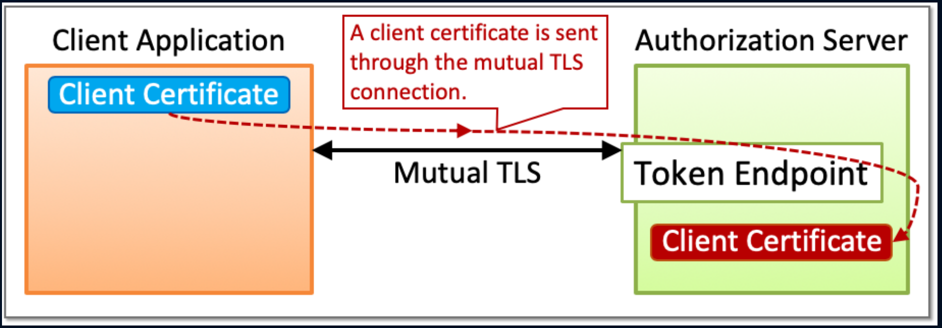

```
eyJhbGciOiJSUzI1NiJ9.eyJhdCI6ICI4dXloZ3Q2Nzg5MDQ5ZGFmc2RmMjM0ZzMiLCJ0cyI6IDMx
NjUzODMsImh0dHAiOnsidiI6IlBPU1QiLCJ1IjoibG9jYWhvc3Q6OTAwMiJ9fQo.m2Na5CCbyt
0bvmiWIgWB_yJ5ETsmrB5uB_hMu7a_bWqn8UoLZxadN8s9joIgfzVO9vl757DvMPFDiE2XWw1m
rfIKn6Epqjb5xPXxqcSJEYoJ1bkbIP1UQpHy8VRpvMcM1JB3LzpLUfe6zhPBxnnO4axKgcQE8Sl
gXGvGAsPqcct92Xb76G04q3cDnEx_hxXO8XnUl2pniKW2C2vY4b5Yyqu-mrXb6r2F4YkTkrkHH
GoFH4w6phIRv3Ku8Gm1_MwhiIDAKPz3_1rRVP_jkID9R4osKZOeBRcosVEW3MoPqcEL2OXRrLh
Yjj9XMdXo8ayjz_6BaRI0VUW3RDuWHP9Dmg
```


  - Finally, the token is sent to resource provider, in the Authorization header

```
HTTP POST /foo
Host: example.org
Authorization: PoP eyJhbGciOiJSUzI1NiJ9.eyJhdCI6ICI4dXloZ3Q2Nzg5MDQ5...
```

PoP – Another way

- **To avoid the key generation and transmission**
  - We can use the Mutual TLS authentication feature and have a client certificate and private key on the app side



  - The server verifies the certificate and extracts the public key that it binds to the token
  - The client uses the private key to sign the token
    - The resource provider also receives the same certificate, and use it to verify the token
  - A disadvantage could be the use of same key for several tokens
    - Can be mitigated if the app server, AuthZ server, and resource provider share and trust the same private CA
    - Make the client app generate a new certificate for each token it obtains

Web applications common attacks

- **OWASP lists the top 10 web apps vulnerabilities and attacks**
  - The list is periodically renewed
  - https://www.owasp.org/index.php/Category:OWASP_Top_Ten_Project
  - Complete characterization and countermeasures are included

OWASP Top 10 - 2013	→	OWASP Top 10 - 2017
A1 – Injection	→	A1:2017-Injection
A2 – Broken Authentication and Session Management	→	A2:2017-Broken Authentication
A3 – Cross-Site Scripting (XSS)	↘	A3:2017-Sensitive Data Exposure
A4 – Insecure Direct Object References [Merged+A7]	∪	A4:2017-XML External Entities (XXE) [NEW]
A5 – Security Misconfiguration	↘	A5:2017-Broken Access Control [Merged]
A6 – Sensitive Data Exposure	↗	A6:2017-Security Misconfiguration
A7 – Missing Function Level Access Contr [Merged+A4]	∪	A7:2017-Cross-Site Scripting (XSS)
A8 – Cross-Site Request Forgery (CSRF)	☒	A8:2017-Insecure Deserialization [NEW, Community]
A9 – Using Components with Known Vulnerabilities	→	A9:2017-Using Components with Known Vulnerabilities
A10 – Unvalidated Redirects and Forwards	☒	A10:2017-Insufficient Logging&Monitoring [NEW,Comm.]